Outline

- Graph generalities
- Graph processing
- Storing the graph
- Aside: Variable-length representations
- Graph compression with instantaneous codes: BVGraph
- BVGraph for general graphs: LLPA
- Graph compression with Elias-Fano: EFGraph
Graph generalities
A graph $G = (V_G, E_G)$ is defined by:

- A set $V_G$ of nodes
- A set $E_G \subseteq V_G \times V_G$ of arcs (ordered pairs of nodes).

Other authors call these directed graphs (or digraphs, or networks).

Note that pairs of the form $(x, x)$ (loops) are allowed (many authors don't take loop into consideration).

$G$ is often dropped from the indices, when clear from the context.
A graph $G = (V_G, E_G)$ is defined by:

- A set $V_G$ of nodes
- A set $E_G \subseteq V_G \times V_G$ of arcs (ordered pairs of nodes)

Other authors call these directed graphs (or digraphs, or networks). Note that pairs of the form $\langle x, x \rangle$ (loops) are allowed (many authors don't take loop into consideration).

$G$ is often dropped from the indices, when clear from the context.
A graph $G = (V_G, E_G)$ is defined by:

- A set $V = V_G$ of *nodes*
A graph $G = (V_G, E_G)$ is defined by:

- A set $V = V_G$ of nodes
- A set $E = E_G \subseteq V \times V$ of arcs (ordered pairs of nodes).
A graph $G = (V_G, E_G)$ is defined by:

- A set $V = V_G$ of nodes
- A set $E = E_G \subseteq V \times V$ of arcs (ordered pairs of nodes).

Other authors call these *directed graphs* (or digraphs, or networks).
A graph $G = (V_G, E_G)$ is defined by:

- A set $V = V_G$ of nodes
- A set $E = E_G \subseteq V \times V$ of arcs (ordered pairs of nodes).

Other authors call these directed graphs (or digraphs, or networks). Note that pairs of the form $(x, x)$ (loops) are allowed (many authors don’t take loop into consideration).
A graph $G = (V_G, E_G)$ is defined by:

- A set $V = V_G$ of nodes
- A set $E = E_G \subseteq V \times V$ of arcs (ordered pairs of nodes).

Other authors call these *directed graphs* (or digraphs, or networks). Note that pairs of the form $(x, x)$ (*loops*) are allowed (many authors don’t take loop into consideration). $G$ is often dropped from the indices, when clear from the context.
The transpose $G^T = (V, E^T)$ of $G = (V, E)$ is defined by:

$$E^T = \{(y, x) \mid (x, y) \in E\}.$$
The transpose $G^T = (V, E^T)$ of $G = (V, E)$ is defined by:

$$E^T = \{(y, x) \mid (x, y) \in E\}.$$ 

A graph $G$ is symmetric iff $G = G^T$. 
The transpose $G^T = (V, E^T)$ of $G = (V, E)$ is defined by:

$$E^T = \{(y, x) \mid (x, y) \in E\}.$$ 

A graph $G$ is symmetric iff $G = G^T$.

We similarly define $G^s = (V, E \cup E^T)$ (the symmetric closure of $G$).
The transpose $G^T = (V, E^T)$ of $G = (V, E)$ is defined by:

$$E^T = \{(y, x) \mid (x, y) \in E\}.$$ 

A graph $G$ is symmetric iff $G = G^T$.

We similarly define $G^s = (V, E \cup E^T)$ (the symmetric closure of $G$).
The transpose $G^T = (V, E^T)$ of $G = (V, E)$ is defined by:

$$E^T = \{(y, x) \mid (x, y) \in E\}.$$ 

A graph $G$ is symmetric iff $G = G^T$.

We similarly define $G^s = (V, E \cup E^T)$ (the symmetric closure of $G$).

- Undirected graphs can be safely identified with symmetric graphs.
The transpose $G^T = (V, E^T)$ of $G = (V, E)$ is defined by:

$$E^T = \{(y, x) \mid (x, y) \in E\}.$$ 

A graph $G$ is symmetric iff $G = G^T$. We similarly define $G^s = (V, E \cup E^T)$ (the symmetric closure of $G$).

- Undirected graphs can be safely identified with symmetric graphs.
- In an undirected graphs, nodes are often called vertices and pairs of opposite arcs are called edges.
In some applications, one may want more than one arc between two nodes (i.e., that $E$ is a multiset of pairs, instead of a set). We call these generalization *multigraphs*.
In some applications, one may want more than one arc between two nodes (i.e., that $E$ is a multiset of pairs, instead of a set). We call these generalization *multigraphs*.

In some other applications, $E$ is not a set of pairs, but a set of $r$-tuples. In this case, we talk of *hypergraphs*. 
A graph can be labelled on its nodes and/or on its arcs. Node-labelling functions map nodes (or arcs) to a set of suitable labels.
A graph can be labelled on its nodes and/or on its arcs. Node-labelling functions map nodes (or arcs) to a set of suitable labels. In the case of undirected graphs one usually requires that \((x, y)\) has the same label as \((y, x)\), so that one can think of labels being assigned to edges.
A graph can be labelled on its nodes and/or on its arcs. Node-labelling functions map nodes (or arcs) to a set of suitable labels.

In the case of undirected graphs one usually requires that \((x, y)\) has the same label as \((y, x)\), so that one can think of labels being assigned to edges.

A special case of labelling is the assignment of real values, that is often called a \textit{weighting function} (hence we call a graph node-weighted or arc-weighted).
Graphs (of some kind) pop up everywhere. Examples:
Graphs everywhere...

Graphs (of some kind) pop up everywhere. Examples:

- social networks, either directed (Twitter) or undirected (Facebook)
Graphs everywhere... 

Graphs (of some kind) pop up everywhere. Examples:

- social networks, either directed (Twitter) or undirected (Facebook)
- biological systems (gene interaction networks, protein networks)
Graphs (of some kind) pop up everywhere. Examples:

- social networks, either directed (Twitter) or undirected (Facebook)
- biological systems (gene interaction networks, protein networks)
- web graphs
Graphs (of some kind) pop up everywhere. Examples:

- social networks, either directed (Twitter) or undirected (Facebook)
- biological systems (gene interaction networks, protein networks)
- web graphs
- Internet autonomous systems
Graphs (of some kind) pop up everywhere. Examples:

- social networks, either directed (Twitter) or undirected (Facebook)
- biological systems (gene interaction networks, protein networks)
- web graphs
- Internet autonomous systems
- semantic networks, knowledge graphs
Graphs (of some kind) pop up everywhere. Examples:

- social networks, either directed (Twitter) or undirected (Facebook)
- biological systems (gene interaction networks, protein networks)
- web graphs
- Internet autonomous systems
- semantic networks, knowledge graphs
- collaboration graphs
Graphs everywhere...

Graphs (of some kind) pop up everywhere. Examples:

- social networks, either directed (Twitter) or undirected (Facebook)
- biological systems (gene interaction networks, protein networks)
- web graphs
- Internet autonomous systems
- semantic networks, knowledge graphs
- collaboration graphs
- citation networks
Graphs (of some kind) pop up everywhere. Examples:

- social networks, either directed (Twitter) or undirected (Facebook)
- biological systems (gene interaction networks, protein networks)
- web graphs
- Internet autonomous systems
- semantic networks, knowledge graphs
- collaboration graphs
- citation networks
- …
These graphs may have a *humongous* number of vertices (not rarely, they have billions of nodes!).
These graphs may have a *humongous* number of vertices (not rarely, they have billions of nodes!). Typically, though, they are very *sparse*:
These graphs may have a *humongous* number of vertices (not rarely, they have billions of nodes!). Typically, though, they are very *sparse*: A *sparse graph* is one with $O(n)$ arcs (instead of $O(n^2)$).
A path in $G$ is a sequence $\pi = x_0, x_1, \ldots, x_k \in V$ such that $(x_i, x_{i+1}) \in E$ for all $i = 0, \ldots, k - 1$. We say that:

- $\pi$ starts at node $x_0$ (also called the source of $\pi$)
- $\pi$ ends at node $x_k$ (also called the target of $\pi$)
- $\pi$ has length $|\pi| = k$
- $\pi$ is simple if $x_0, \ldots, x_{k-1}$ are all distinct
- $\pi$ is a cycle if $k > 0$ and the source and target coincide.

If there is a path from $x$ to $y$ we say that $y$ is reachable from $x$. If there is a cycle, $G$ is called cyclic.
A path in $G$ is a sequence $\pi = x_0, x_1, \ldots, x_k \in V$ such that $(x_i, x_{i+1}) \in E$ for all $i = 0, \ldots, k - 1$. We say that:

- $\pi$ starts at node $x_0$ (also called the source of $\pi$)
- $\pi$ ends at node $x_k$ (also called the target of $\pi$)
- $\pi$ has length $|\pi| = k$
- $\pi$ is simple if $x_0, \ldots, x_{k-1}$ are all distinct
- $\pi$ is a cycle iff $k > 0$ and the source and target coincide.

If there is a path from $x$ to $y$ we say that $y$ is reachable from $x$ if there is a cycle, $G$ is called cyclic.
A path in $G$ is a sequence $\pi = x_0, x_1, \ldots, x_k \in V$ such that $(x_i, x_{i+1}) \in E$ for all $i = 0, \ldots, k - 1$. We say that:

- $\pi$ starts at node $x_0$ (also called the source of $\pi$)
A path in $G$ is a sequence $\pi = x_0, x_1, \ldots, x_k \in V$ such that $(x_i, x_{i+1}) \in E$ for all $i = 0, \ldots, k - 1$. We say that:

- $\pi$ starts at node $x_0$ (also called the source of $\pi$)
- $\pi$ ends at node $x_k$ (also called the target of $\pi$)

if there is a path from $x$ to $y$ we say that $y$ is reachable from $x$ if there is a cycle, $G$ is called cyclic.
A path in \( G \) is a sequence \( \pi = x_0, x_1, \ldots, x_k \in V \) such that \((x_i, x_{i+1}) \in E\) for all \( i = 0, \ldots, k - 1 \). We say that:

- \( \pi \) starts at node \( x_0 \) (also called the source of \( \pi \))
- \( \pi \) ends at node \( x_k \) (also called the target of \( \pi \))
- has length \( |\pi| = k \)
A path in $G$ is a sequence $\pi = x_0, x_1, \ldots, x_k \in V$ such that $(x_i, x_{i+1}) \in E$ for all $i = 0, \ldots, k - 1$. We say that:

- $\pi$ starts at node $x_0$ (also called the source of $\pi$)
- $\pi$ ends at node $x_k$ (also called the target of $\pi$)
- has length $|\pi| = k$
- is simple if $x_0, \ldots, x_{k-1}$ are all distinct
A path in $G$ is a sequence $\pi = x_0, x_1, \ldots, x_k \in V$ such that $(x_i, x_{i+1}) \in E$ for all $i = 0, \ldots, k - 1$. We say that:

- $\pi$ starts at node $x_0$ (also called the source of $\pi$)
- $\pi$ ends at node $x_k$ (also called the target of $\pi$)
- has length $|\pi| = k$
- is simple if $x_0, \ldots, x_{k-1}$ are all distinct
- it is a cycle iff $k > 0$ and the source and target coincide.
A path in $G$ is a sequence $\pi = x_0, x_1, \ldots, x_k \in V$ such that $(x_i, x_{i+1}) \in E$ for all $i = 0, \ldots, k - 1$. We say that:

- $\pi$ starts at node $x_0$ (also called the source of $\pi$)
- $\pi$ ends at node $x_k$ (also called the target of $\pi$)
- has length $|\pi| = k$
- is simple if $x_0, \ldots, x_{k-1}$ are all distinct
- it is a cycle iff $k > 0$ and the source and target coincide.
- if there is a path from $x$ to $y$ we say that $y$ is reachable from $x$
A path in $G$ is a sequence $\pi = x_0, x_1, \ldots, x_k \in V$ such that $(x_i, x_{i+1}) \in E$ for all $i = 0, \ldots, k - 1$. We say that:

- $\pi$ starts at node $x_0$ (also called the source of $\pi$)
- $\pi$ ends at node $x_k$ (also called the target of $\pi$)
- has length $|\pi| = k$
- is simple if $x_0, \ldots, x_{k-1}$ are all distinct
- it is a cycle iff $k > 0$ and the source and target coincide.
- if there is a path from $x$ to $y$ we say that $y$ is reachable from $x$
- if there is a cycle, $G$ is called cyclic.
Strongly connected components

Let \( x \approx y \) iff there is a path from \( x \) to \( y \) and vice-versa. The equivalence classes of \( \approx \) are called the *strongly connected components* (SCCs) of \( G \). The SCCs of \( G^s \) are called the *weakly connected components* (WCCs) of \( G \): in the case of a symmetric graph, WCCs and SCCs coincide (and we just talk of “connected components”).
Let \( x \approx y \) iff there is a path from \( x \) to \( y \) and vice-versa. The equivalence classes of \( \approx \) are called the \textit{strongly connected components} (SCCs) of \( G \). The SCCs of \( G^s \) are called the \textit{weakly connected components} (WCCs) of \( G \): in the case of a symmetric graph, WCCs and SCCs coincide (and we just talk of “connected components”).
The *reduced graph* $G^\dagger$ is the graph whose nodes are the SCCs of $G$, with an arc from $[x]$ to $[y]$ whenever there is a node $x' \approx x$ and a node $y' \approx y$ with $(x', y') \in E$. Theorem $G^\dagger$ is an acyclic graph.
The *reduced graph* $G^\dagger$ is the graph whose nodes are the SCCs of $G$, with an arc from $[x]$ to $[y]$ whenever there is a node $x' \approx x$ and a node $y' \approx y$ with $(x', y') \in E$.

**Theorem**

$G^\dagger$ is an acyclic graph.
Strongly connected components

The *reduced graph* $G^\dagger$ is the graph whose nodes are the SCCs of $G$, with an arc from $[x]$ to $[y]$ whenever there is a node $x' \approx x$ and a node $y' \approx y$ with $(x', y') \in E$.

**Theorem**

$G^\dagger$ is an acyclic graph.
Given $G = (V, E)$ and $x \in V$, define:

- $N^{-G}(x) = \{y \mid (y, x) \in E\}$ (in-neighborhood of $x$, predecessors of $x$)
- $N^{+G}(x) = \{y \mid (x, y) \in E\}$ (out-neighborhood of $x$, successors of $x$)
- $d^{-G}(x) = \left| N^{-G}(x) \right|$ (in-degree of $x$)
- $d^{+G}(x) = \left| N^{+G}(x) \right|$ (out-degree of $x$)
Given $G = (V, E)$ and $x \in V$, define:

- $N_G^-(x) = \{y \mid (y, x) \in E\}$ (in-neighborhood of $x$, predecessors of $x$)
- $N_G^+(x) = \{y \mid (x, y) \in E\}$ (out-neighborhood of $x$, successors of $x$)
- $d_G^-(x) = |N_G^-(x)|$ (in-degree of $x$)
- $d_G^+(x) = |N_G^+(x)|$ (out-degree of $x$)
Neighborhoods and degrees

Given $G = (V, E)$ and $x \in V$, define:

- $N^-_G(x) = \{y \mid (y, x) \in E\}$ (in-neighborhood of $x$, predecessors of $x$)
- $N^+_G(x) = \{y \mid (x, y) \in E\}$ (out-neighborhood of $x$, successors of $x$)

$d^-_G(x) = |N^-_G(x)|$ (in-degree of $x$)

$d^+_G(x) = |N^+_G(x)|$ (out-degree of $x$)
Given $G = (V, E)$ and $x \in V$, define:

- $N^-_G(x) = \{y \mid (y,x) \in E\}$ (in-neighborhood of $x$, predecessors of $x$)
- $N^+_G(x) = \{y \mid (x,y) \in E\}$ (out-neighborhood of $x$, successors of $x$)
- $d^-_G(x) = |N^-_G(x)|$ (in-degree of $x$)
Given $G = (V, E)$ and $x \in V$, define:

- $N_G^-(x) = \{ y \mid (y, x) \in E \}$ (in-neighborhood of $x$, predecessors of $x$)
- $N_G^+(x) = \{ y \mid (x, y) \in E \}$ (out-neighborhood of $x$, successors of $x$)
- $d_G^-(x) = |N_G^-(x)|$ (in-degree of $x$)
- $d_G^+(x) = |N_G^+(x)|$ (out-degree of $x$)
Given $G = (V, E)$ and $x \in V$, define:

\[
\text{in-directed clustering coefficient of } x : \quad c^{-G}(x) = \frac{|E_G \cap (N_G(x) \times N_G(x))|}{d_G(x)}
\]

or, if loop are not allowed:

\[
\text{in-directed clustering coefficient of } x : \quad c^{-G}(x) = \frac{|E_G \cap (N_G(x) \times N_G(x))|}{d_G(x) \cdot (d_G(x) - 1)}
\]

$c^+_G(x)$ is defined similarly for undirected loopless graphs:

\[
\text{in-directed clustering coefficient of } x : \quad c^+_G(x) = \frac{2|E_G \cap (N_G(x) \times N_G(x))|}{d_G(x) \cdot (d_G(x) - 1)}
\]
(Local) clustering coefficient

Given $G = (V, E)$ and $x \in V$, define:

- in-directed clustering coefficient of $x$:

$$c_G^-(x) = \frac{|E_G \cap (N_G^- (x) \times N_G^- (x))|}{d_G^-(x)^2}$$

or, if loop are not allowed:

$$c_G^-(x) = \frac{|E_G \cap (N_G^- (x) \times N_G^- (x))|}{d_G^-(x) \cdot (d_G^-(x) - 1)}$$
(Local) clustering coefficient

Given $G = (V, E)$ and $x \in V$, define:

- in-directed clustering coefficient of $x$:

$$c_{G}^{-}(x) = \frac{|E_G \cap (N_G^{-}(x) \times N_G^{-}(x))|}{d_{G}^{-}(x)^2}$$

or, if loop are not allowed:

$$c_{G}^{-}(x) = \frac{|E_G \cap (N_G^{-}(x) \times N_G^{-}(x))|}{d_{G}^{-}(x) \cdot (d_{G}^{-}(x) - 1)}$$

- $c_{G}^{+}(x)$ is defined similarly
Given $G = (V, E)$ and $x \in V$, define:

- in-directed clustering coefficient of $x$:

$$c_G^-(x) = \frac{|E_G \cap (N_G^-(x) \times N_G^-(x))|}{d_G^-(x)^2}$$

or, if loop are not allowed:

$$c_G^-(x) = \frac{|E_G \cap (N_G^-(x) \times N_G^-(x))|}{d_G^-(x) \cdot (d_G^-(x) - 1)}$$

- $c_G^+(x)$ is defined similarly
- for undirected loopless graphs:

$$c_G(x) = \frac{2|E_G \cap (N_G(x) \times N_G(x))|}{d_G(x) \cdot (d_G(x) - 1)}$$
A graph \textit{morphism} $f : G \rightarrow H$ is a function $f : V_G \rightarrow V_H$ such that $(x, y) \in E_G$ if and only if $(f(x), f(y)) \in E_H$. A bijective graph morphism is called an \textit{isomorphism}. If there exists an isomorphism between $G$ and $H$ we say that $G$ and $H$ are \textit{isomorphic}, and write $G \cong H$. 

Graph generalities
A graph morphism $f : G \to H$ is a function $f : V_G \to V_H$ such that $(x, y) \in E_G$ if and only if $(f(x), f(y)) \in E_H$.

A bijective graph morphism is called an isomorphism. If there exists an isomorphism between $G$ and $H$ we say that $G$ and $H$ are isomorphic, and write $G \cong H$. 
Graph processing
A graph property is a function that associates a value to each graph.
A graph property is a function that associates a value to each graph.

- Binary graph properties
Graph property

A *graph property* is a function that associates a value to each graph.

- Binary graph properties
  - Is the graph planar?
Graph property

A *graph property* is a function that associates a value to each graph.

- **Binary graph properties**
  - Is the graph planar?
  - Is the graph connected?
A graph property is a function that associates a value to each graph.

- Binary graph properties
  - Is the graph planar?
  - Is the graph connected?

- Scalar properties
A graph property is a function that associates a value to each graph.

- **Binary graph properties**
  - Is the graph planar?
  - Is the graph connected?

- **Scalar properties**
  - Average degree
A *graph property* is a function that associates a value to each graph.

- **Binary graph properties**
  - Is the graph planar?
  - Is the graph connected?
- **Scalar properties**
  - Average degree
  - Average (shortest-path) distance
A *graph property* is a function that associates a value to each graph.

- **Binary graph properties**
  - Is the graph planar?
  - Is the graph connected?

- **Scalar properties**
  - Average degree
  - Average (shortest-path) distance
  - Average clustering coefficient
A *graph property* is a function that associates a value to each graph.

- **Binary graph properties**
  - Is the graph planar?
  - Is the graph connected?

- **Scalar properties**
  - Average degree
  - Average (shortest-path) distance
  - Average clustering coefficient

- **Distributions**
A *graph property* is a function that associates a value to each graph.

- **Binary graph properties**
  - Is the graph planar?
  - Is the graph connected?

- **Scalar properties**
  - Average degree
  - Average (shortest-path) distance
  - Average clustering coefficient

- **Distributions**
  - Degree distribution
A \textit{graph property} is a function that associates a value to each graph.

- **Binary graph properties**
  - Is the graph planar?
  - Is the graph connected?

- **Scalar properties**
  - Average degree
  - Average (shortest-path) distance
  - Average clustering coefficient

- **Distributions**
  - Degree distribution
  - Distance distribution
A graph property is a function that associates a value to each graph.

- **Binary graph properties**
  - Is the graph planar?
  - Is the graph connected?

- **Scalar properties**
  - Average degree
  - Average (shortest-path) distance
  - Average clustering coefficient

- **Distributions**
  - Degree distribution
  - Distance distribution

- **Vector properties**
A *graph property* is a function that associates a value to each graph.

- **Binary graph properties**
  - Is the graph planar?
  - Is the graph connected?

- **Scalar properties**
  - Average degree
  - Average (shortest-path) distance
  - Average clustering coefficient

- **Distributions**
  - Degree distribution
  - Distance distribution

- **Vector properties**
  - Local clustering coefficient
A graph property is a function that associates a value to each graph.

- **Binary graph properties**
  - Is the graph planar?
  - Is the graph connected?

- **Scalar properties**
  - Average degree
  - Average (shortest-path) distance
  - Average clustering coefficient

- **Distributions**
  - Degree distribution
  - Distance distribution

- **Vector properties**
  - Local clustering coefficient
  - Centrality (e.g., eccentricity)
A graph property $P$ is *isomorphism-invariant* iff

$$G \cong H \text{ implies } P(G) = P(H).$$
A graph property $P$ is isomorphism-invariant iff

$$G \cong H \text{ implies } P(G) = P(H).$$

Properties that are not isomorphism-invariant are tricky (they depend on the specific identity of nodes).
A graph property $P$ is \textit{isomorphism-invariant} iff

$$G \cong H \text{ implies } P(G) = P(H).$$

Properties that are \textit{not} isomorphism-invariant are tricky (they depend on the specific identity of nodes). If we limit ourselves to isomorphism-invariant properties, we can assume \textit{w.l.o.g.} that $V_G = \{0, 1, \ldots, n - 1\}$. 
If you store a graph $G$ (in some way), you can access it through different primitives, such as...

- **Direct access queries:**
  - $G.arc(x,y)$ (is there an arc from $x$ to $y$ in $G$?)
  - $G.d\pm(x)$ (what is the in/out-degree of $x$ in $G$?)
  - $G.N\pm(x)$ (enumerate the in/out-neighbors of $x$ in $G$; possibly in order of id)

- **Sequential access (a.k.a., streaming):**
  - $G.E$ (enumerate the arcs, possibly preserving the consecutivity of in/out-neighborhoods)

If $G.E$ can be called only once (or $O(1)$ times), the access is "streaming."
Accessing a graph

If you store a graph $G$ (in some way), you can access it through different primitives, such as...

- direct access queries:
  - $G.a(x, y)$ (is there an arc from $x$ to $y$ in $G$?)
  - $G.d\pm(x)$ (what is the in/out-degree of $x$ in $G$?)
  - $G.N\pm(x)$ (enumerate the in/out-neighbors of $x$ in $G$; possibly in order of id)

sequential access (a.k.a., streaming):
- $G.E$ (enumerate the arcs, possibly preserving the consecutivity of in/out-neighborhoods)

if $G.E$ can be called only once (or $O(1)$ times), the access is "streaming."
If you store a graph $G$ (in some way), you can access it through different primitives, such as...

- direct access queries:
  - $G.arc(x, y)$ (is there an arc from $x$ to $y$ in $G$?)

- sequential access (a.k.a., streaming):
  - $G.E$ (enumerate the arcs, possibly preserving the consecutivity of in/out-neighborhoods)
  - if $G.E$ can be called only once (or $O(1)$ times), the access is "streaming."
Accessing a graph

If you store a graph $G$ (in some way), you can access it through different primitives, such as...

- **direct access queries:**
  - $G.\text{arc}(x, y)$ (is there an arc from $x$ to $y$ in $G$?)
  - $G.d^\pm(x)$ (what is the in/out-degree of $x$ in $G$?)

- **sequential access (a.k.a., streaming):**
  - $G.E$ (enumerate the arcs, possibly preserving the consecutivity of in/out-neighborhoods)
If you store a graph $G$ (in some way), you can access it through different primitives, such as... 

- **direct access queries:**
  - $G.arc(x, y)$ (is there an arc from $x$ to $y$ in $G$?)
  - $G.d^\pm(x)$ (what is the in/out-degree of $x$ in $G$?)
  - $G.N^\pm(x)$ (enumerate the in/out-neighbors of $x$ in $G$; possibly in order of id)
If you store a graph $G$ (in some way), you can access it through different primitives, such as...

- **direct access queries:**
  - $G.arc(x, y)$ (is there an arc from $x$ to $y$ in $G$?)
  - $G.d^\pm(x)$ (what is the in/out-degree of $x$ in $G$?)
  - $G.N^\pm(x)$ (enumerate the in/out-neighbors of $x$ in $G$; possibly in order of id)

- **sequential access** (a.k.a., streaming):
Accessing a graph

If you store a graph $G$ (in some way), you can access it through different primitives, such as...

- **direct access queries:**
  - $G.arc(x, y)$ (is there an arc from $x$ to $y$ in $G$?)
  - $G.d^\pm(x)$ (what is the in/out-degree of $x$ in $G$?)
  - $G.N^\pm(x)$ (enumerate the in/out-neighbors of $x$ in $G$; possibly in order of id)

- **sequential access (a.k.a., streaming):**
  - $G.E$ (enumerate the arcs, possibly preserving the consecutivity of in/out-neighborhoods)
If you store a graph $G$ (in some way), you can access it through different primitives, such as...

- **direct access queries:**
  - $G.arc(x, y)$ (is there an arc from $x$ to $y$ in $G$?)
  - $G.d^\pm(x)$ (what is the in/out-degree of $x$ in $G$?)
  - $G.N^\pm(x)$ (enumerate the in/out-neighbors of $x$ in $G$; possibly in order of id)

- **sequential access (a.k.a., streaming):**
  - $G.E$ (enumerate the arcs, possibly preserving the consecutivity of in/out-neighborhoods)
  - if $G.E$ can be called only once (or $O(1)$ times), the access is “streaming”.
Computing graph properties

Typical problem:
Computing graph properties

Typical problem:

- for a given property $P$
Typical problem:
- for a given property $P$
- for a given access mode
Computing graph properties

Typical problem:
- for a given property $P$
- for a given access mode
- write an algorithm that computes (or approximates, in some sense) $G \mapsto P(G)$
Storing the graph
What do we mean by “storing the graph”? 

- Having a data structure that allows you, for a given node, to know its successors.
What do we mean by... "storing the graph"?

- Having a data structure that allows you, for a given node, to know its successors.
- If the graph is node-labelled (e.g., a web graph with URLs as node labels): having a way to know which label corresponds to a given node and vice versa.
What do we mean by... "storing the graph"?

- Having a data structure that allows you, for a given node, to know its successors.

- If the graph is node-labelled (e.g., a web graph with URLs as node labels): having a way to know which label corresponds to a given node and vice versa.

Here: we only consider the former problem, not the latter!
Information-theoretical lower bound

How much space do we need to store a graph with $n$ nodes and $m$ arcs?

Not less than $\log(n^2m) \approx m \log(n^2m) + O(m)$ under the hypothesis that $m = o(n^2)$.

This means about $\log(n/d) + O(1)$ bits per arc. But complex networks are NOT random graphs!
Information-theoretical lower bound

How much space do we need to store a graph with $n$ nodes and $m$ arcs? Not less than

$$\log \binom{n^2}{m} \approx m \log \left( \frac{n^2}{m} \right) + O(m)$$

under the hypothesis that $m = o\left(\frac{n^2}{d}\right)$, where $d = \frac{m}{n}$ is the average degree.

This means about $\log(\frac{n}{d}) + O(1)$ bits per arc. But complex networks are NOT random graphs!
How much space do we need to store a graph with $n$ nodes and $m$ arcs? Not less than

$$\log \binom{n^2}{m} \approx m \log \left( \frac{n^2}{m} \right) + O(m)$$

under the hypothesis that $m = o(n^2)$

$$m \log \left( \frac{n}{d} \right) + O(m)$$

where $d = m/n$ is the average degree.
How much space do we need to store a graph with \( n \) nodes and \( m \) arcs? Not less than

\[
\log \binom{n^2}{m} \approx m \log \left( \frac{n^2}{m} \right) + O(m)
\]

under the hypothesis that \( m = o(n^2) \)

\[
m \log \left( \frac{n}{d} \right) + O(m)
\]

where \( d = m/n \) is the average degree. This means about \( \log(n/d) + O(1) \) bits per arc. But *complex networks are NOT random graphs!*.
Naive graph representations (1)

Most popular naive representations for a graph $G$:

- **Adjacency matrix**: An $n \times n$ binary matrix $G$ with $G_{xy} = 1$ iff $(x, y) \in E_G$.
  - **Features**: It occupies $n^2$ bits (i.e., $n^2/m = n^2/d$ bits per arc), exponentially more than the information-theoretic lower bound $\log n/d$.
  - A relation $(x, y)$ takes constant time; enumerating the neighborhoods takes time $O(n)$; scanning the graph sequentially takes time $O(n^2)$, highly unsuitable for sparse graphs!
Most popular naive representations for a graph $G$:

- **adjacency matrix**: a $n \times n$ binary matrix $G$ with $G_{xy} = 1$ iff $(x, y) \in E_G$. 

  - Features:
    - *it occupies* $n^2$ bits (i.e., $n^2 / m = n^2 / nd$ bits per arc: exponentially more than the information-theoretic lower bound $\log n / d$).
    - $G$'s arc $(x, y)$ takes constant time:
    - Enumerating the neighborhoods takes time $O(n)$.
    - Scanning the graph sequentially takes time $O(n^2)$.

  - Highly unsuitable for sparse graphs!
Most popular naive representations for a graph $G$:

- **adjacency matrix**: a $n \times n$ binary matrix $G$ with $G_{xy} = 1$ iff $(x, y) \in E_G$.

Features:

- occupies $n^2$ bits (i.e., $n^2/m = n^2/nd = n/d$ bits per arc: exponentially more than the information-theoretic lower bound $\log n/d$).
- arc $(x, y)$ takes constant time:
- enumerating the neighborhoods takes time $O(n)$:
- scanning the graph sequentially takes time $O(n^2)$.

Highly unsuitable for sparse graphs!
Most popular naive representations for a graph $G$:

- **adjacency matrix**: a $n \times n$ binary matrix $G$ with $G_{xy} = 1$ iff $(x, y) \in E_G$.

Features:

- it occupies $n^2$ bits (i.e., $n^2/m = n^2/nd = n/d$ bits per arc: exponentially more than the information-theoretic lower bound $\log n/d$)
Most popular naive representations for a graph $G$:

- **adjacency matrix**: a $n \times n$ binary matrix $G$ with $G_{xy} = 1$ iff $(x, y) \in E_G$.

Features:

- it occupies $n^2$ bits (i.e., $n^2/m = n^2/nd = n/d$ bits per arc: exponentially more than the information-theoretic lower bound $\log n/d$)
- $G.arc(x, y)$ takes constant time:
Most popular naive representations for a graph $G$:

- adjacency matrix: a $n \times n$ binary matrix $G$ with $G_{xy} = 1$ iff $(x, y) \in E_G$.

Features:

- it occupies $n^2$ bits (i.e., $n^2/m = n^2/nd = n/d$ bits per arc: exponentially more than the information-theoretic lower bound $\log n/d$)
- $G.arc(x, y)$ takes constant time:
- enumerating the neighborhoods takes time $O(n)$
Most popular naive representations for a graph $G$:

- **adjacency matrix**: a $n \times n$ binary matrix $G$ with $G_{xy} = 1$ iff $(x, y) \in E_G$.

**Features**:

- it occupies $n^2$ bits (i.e., $n^2/m = n^2/nd = n/d$ bits per arc: exponentially more than the information-theoretic lower bound $\log n/d$)
- $G.arc(x, y)$ takes constant time:
- enumerating the neighborhoods takes time $O(n)$
- scanning the graph sequentially takes time $O(n^2)$
Most popular naive representations for a graph $G$:

- **adjacency matrix**: a $n \times n$ binary matrix $G$ with $G_{xy} = 1$ iff $(x, y) \in E_G$.

**Features:**

- It occupies $n^2$ bits (i.e., $n^2/m = n^2/nd = n/d$ bits per arc: exponentially more than the information-theoretic lower bound $\log n/d$).
- $G.arc(x, y)$ takes constant time:
- Enumerating the neighborhoods takes time $O(n)$.
- Scanning the graph sequentially takes time $O(n^2)$.
- **Highly unsuitable for sparse graphs!**
Second most popular naive representations for a graph $G$:

- **Adjacency lists:** one list per node, containing its successors (in increasing order).
  - Features:
    - Memory occupied: see below.
    - $G(x, y)$ takes $O(d)$ time.
    - Enumerating the neighborhoods takes time $O(d)$.
    - Scanning the graph sequentially takes time $O(m)$.
Second most popular naive representations for a graph $G$:

- adjacency lists: one list per node, containing its successors (in increasing order).
Second most popular naive representations for a graph $G$:

- adjacency lists: one list per node, containing its successors (in increasing order).

Features:
Second most popular naive representations for a graph $G$:

- adjacency lists: one list per node, containing its successors (in increasing order).

Features:

- memory occupied: see below
Second most popular naive representations for a graph $G$:

- adjacency lists: one list per node, containing its successors (in increasing order).

Features:

- memory occupied: see below
- $G.arc(x, y)$ takes $O(d)$ time
Second most popular naive representations for a graph $G$:

- adjacency lists: one list per node, containing its successors (in increasing order).

Features:

- memory occupied: see below
- $G.arc(x, y)$ takes $O(d)$ time
- enumerating the neighborhoods takes time $O(d)$
Second most popular naive representations for a graph $G$:

- adjacency lists: one list per node, containing its successors (in increasing order).

Features:

- memory occupied: see below
- $G.arc(x, y)$ takes $O(d)$ time
- enumerating the neighborhoods takes time $O(d)$
- scanning the graph sequentially takes time $O(m)$
The offset vector tells, for each given node $x$, where the successor list of $x$ starts from. Implicitly, it also gives the degree of each node.
How much space does this representation take?

- **Successor array**: \( m \) elements (arcs), each containing a node (\( \log n \) bits); with 32 bits, we can store up to 4 billion nodes (half of it, if we don’t have unsigned types)
Naive representation

How much space does this representation take?

- **Successor array**: \( m \) elements (arcs), each containing a node (\( \log n \) bits); with 32 bits, we can store up to 4 billion nodes (half of it, if we don’t have unsigned types)
- **Offset array**: \( n \) elements (nodes), each containing an index in the successor array (\( \log m \) bits); with 32 bits, we can store up to 4 billion arcs.
Naive representation

How much space does this representation take?

- **Successor array**: $m$ elements (arcs), each containing a node ($\log n$ bits); with 32 bits, we can store up to 4 billion nodes (half of it, if we don’t have unsigned types)

- **Offset array**: $n$ elements (nodes), each containing an index in the successor array ($\log m$ bits); with 32 bits, we can store up to 4 billion arcs.

All in all, $32(n + m)$ bits. If we assume $m = 8n$ (a very modest assumption on the outdegree), we need $288n$ bits, i.e., 288 bits/node, 36 bits/arc.

We show how to reduce this of an order of magnitude.
Use a variable-length representation for successors. Such a representation should obviously...

- be instantaneously decodable
Use a variable-length representation for successors. Such a representation should obviously...

- be instantaneously decodable
- minimize the expected bitlength.
Idea

Use a variable-length representation for successors. Such a representation should obviously...

- be instantaneously decodable
- minimize the expected bitlength.

What about the offset array?
Use a variable-length representation for successors. Such a representation should obviously...

- be instantaneously decodable
- minimize the expected bitlength.

What about the offset array?

- bit displacement vs. byte displacement (with alignment)
Use a variable-length representation for successors. Such a representation should obviously...

- be instantaneously decodable
- minimize the expected bitlength.

What about the offset array?
- bit displacement vs. byte displacement (with alignment)
- we have to keep an explicit representation of the node degrees (e.g., in the successor array, before every successor list).
Node degrees (blue background), followed by successors. Each number is represented using an instantaneous code (possibly, different for degree and successors).
Aside: Variable-length representations
An instantaneous (binary) code for the set $S$ is a function $c: S \rightarrow \{0, 1\}^*$ such that, for all $x, y \in S$, if $c(x)$ is a prefix of $c(y)$, then $x = y$. 

Aside: Variable-length representations
An *instantaneous (binary) code* for the set $S$ is a function $c : S \rightarrow \{0, 1\}^*$ such that, for all $x, y \in S$, if $c(x)$ is a prefix of $c(y)$, then $x = y$.

Let $l_x$ be the length (in bits) of $c(x)$.
instantaneous code

- An *instantaneous (binary) code* for the set $S$ is a function $c : S \rightarrow \{0, 1\}^*$ such that, for all $x, y \in S$, if $c(x)$ is a prefix of $c(y)$, then $x = y$.

- Let $l_x$ be the length (in bits) of $c(x)$.

- Kraft-McMillan: there exists an instantaneous code with lengths $l_x$ ($x \in S$) if and only if

$$
\sum_{x \in S} 2^{-l_x} \leq 1.
$$
An *instantaneous (binary) code* for the set $S$ is a function $c : S \to \{0, 1\}^*$ such that, for all $x, y \in S$, if $c(x)$ is a prefix of $c(y)$, then $x = y$.

Let $l_x$ be the length (in bits) of $c(x)$.

Kraft-McMillan: there exists an instantaneous code with lengths $l_x$ ($x \in S$) if and only if

$$\sum_{x \in S} 2^{-l_x} \leq 1.$$ 

An instantaneous code for which the equality holds is called “complete”.

Aside: Variable-length representations
Given a set $S$ and an instantaneous code $c : S \to \{0, 1\}^*$, the
expected length of $c$ with respect to some probability distribution
\(p : S \to [0, 1]\) is
\[
E_p[c] = \sum_{s \in S} p(s) \cdot |c(s)|.
\]
Expected length

Given a set $S$ and an instantaneous code $c : S \rightarrow \{0, \}^*$, the \textit{expected length} of $c$ with respect to some probability distribution $p : S \rightarrow [0, 1]$ is

$$E_p[c] = \sum_{s \in S} p(s) \cdot |c(s)|.$$  

• Given a probability distribution $p$, the \textit{optimal code} is the instantaneous code $c_p^* : S \rightarrow \{0, 1\}$ minimizing $E_p[c_p^*]$.  

Aside: Variable-length representations
Expected length

Given a set $S$ and an instantaneous code $c : S \rightarrow \{0, 1\}^*$, the *expected length of $c$* with respect to some probability distribution $p : S \rightarrow [0, 1]$ is

$$E_p[c] = \sum_{s \in S} p(s) \cdot |c(s)|.$$

- Given a probability distribution $p$, the *optimal code* is the instantaneous code $c_p^* : S \rightarrow \{0, 1\}$ minimizing $E_p[c_p^*]$.
- If $S$ is finite:

Aside: Variable-length representations
Expected length

Given a set $S$ and an instantaneous code $c : S \rightarrow \{0, \}^*$, the \textit{expected length} of $c$ with respect to some probability distribution $p : S \rightarrow [0, 1]$ is

$$E_p[c] = \sum_{s \in S} p(s) \cdot |c(s)|.$$  

- Given a probability distribution $p$, the \textit{optimal code} is the instantaneous code $c_p^* : S \rightarrow \{0, 1\}$ minimizing $E_p[c_p^*]$.
- If $S$ is finite:
  - $H[p] = E_p[c_p^*]$ (Shannon’s coding theorem)
Expected length

Given a set $S$ and an instantaneous code $c : S \rightarrow \{0, \}^*$, the expected length of $c$ with respect to some probability distribution $p : S \rightarrow [0, 1]$ is

$$E_p[c] = \sum_{s \in S} p(s) \cdot |c(s)|.$$ 

- Given a probability distribution $p$, the optimal code is the instantaneous code $c_p^* : S \rightarrow \{0, 1\}$ minimizing $E_p[c_p^*]$.
- If $S$ is finite:
  - $H[p] = E_p[c_p^*]$ (Shannon's coding theorem)
  - $c_p^*$ is the Huffman coding.
Given an instantaneous code $c: S \rightarrow \{0, 1\}^*$, define a $p: S \rightarrow [0, 1]$ as $p(x) = 2 - |c(x)|$.

$p$ is called the intended distribution for the code $c$.

To be more precise: $P$ is a probability distribution if $c$ is complete (otherwise it does not sum up to 1, and we need to introduce some normalization factor to turn it into a distribution).

It is easy to see that (if $S$ is finite) $c$ is the optimal code for $p$; in fact:

$$H(p) = \sum_{s \in S} -p(s) \log p(s) = \sum_{s \in S} 2 - |c(s)| - |c(s)| = \sum_{s \in S} p(s) |c(s)| = E_p[c].$$

So, in practice, the choice of the code to use will be based on the expected distribution of the data.

Aside: Variable-length representations
Given an instantaneous code $c : S \rightarrow \{0, 1\}^*$, define a
$p : S \rightarrow [0, 1]$ as

\[ p(x) = 2^{-|c(x)|}. \]
Intended distribution

- Given an instantaneous code \( c : S \rightarrow \{0, 1\}^* \), define a
  \( p : S \rightarrow [0, 1] \) as
  \[
p(x) = 2^{-|c(x)|}.
  \]

- \( p \) is called the intended distribution for the code \( c \).

Aside: Variable-length representations
Given an instantaneous code $c : S \rightarrow \{0, 1\}^*$, define a
$p : S \rightarrow [0, 1]$ as

$$p(x) = 2^{-|c(x)|}.$$

$p$ is called the *intended distribution* for the code $c$.

to be more precise: $P$ is a probability distribution if $c$ is
complete (otherwise it does not sum up to 1, and we need to
introduce some normalization factor to turn it into a
distribution).
Given an instantaneous code \( c : S \rightarrow \{0, 1\}^* \), define a

\( p : S \rightarrow [0, 1] \) as

\[ p(x) = 2^{-|c(x)|}. \]

\( p \) is called the *intended distribution* for the code \( c \).

To be more precise: \( P \) is a probability distribution if \( c \) is complete (otherwise it does not sum up to 1, and we need to introduce some normalization factor to turn it into a distribution).

It is easy to see that (if \( S \) is finite) \( c \) is the optimal code for \( p \); in fact:

\[
H(p) = \sum_{s \in S} -p(s) \log p(s) = \sum_{s \in S} 2^{-|c(s)||c(s)|} = \sum_{s \in S} p(s)|c(s)| = E_p[c].
\]
Intended distribution

- Given an instantaneous code \( c : S \to \{0, 1\}^* \), define a \( p : S \to [0, 1] \) as
  \[
p(x) = 2^{-|c(x)|}.
  \]

- \( p \) is called the intended distribution for the code \( c \).
- To be more precise: \( P \) is a probability distribution if \( c \) is complete (otherwise it does not sum up to 1, and we need to introduce some normalization factor to turn it into a distribution).

It is easy to see that (if \( S \) is finite) \( c \) is the optimal code for \( p \); in fact:

\[
H(p) = \sum_{s \in S} -p(s) \log p(s) = \sum_{s \in S} 2^{-|c(s)|} |c(s)| = \sum_{s \in S} p(s)|c(s)| = E_p[c].
\]

So, in practice, the choice of the code to use will be based on the expected distribution of the data.

Aside: Variable-length representations
If $S = \{1, 2, \ldots, N\}$, to represent an element of $S$ it is sufficient to use $\lceil \log N \rceil$ bits.
Fixed-length coding

- If $S = \{1, 2, \ldots, N\}$, to represent an element of $S$ it is sufficient to use $\lceil \log N \rceil$ bits.
- The fixed-length representation for $S$ uses exactly that number of bits for every element (and represents $x$ using the standard binary coding of $x - 1$ on $\lceil \log N \rceil$ bits).

Aside: Variable-length representations
Fixed-length coding

- If $S = \{1, 2, \ldots, N\}$, to represent an element of $S$ it is sufficient to use $\lceil \log N \rceil$ bits.
- The fixed-length representation for $S$ uses exactly that number of bits for every element (and represents $x$ using the standard binary coding of $x - 1$ on $\lceil \log N \rceil$ bits).
- Intended distribution:
  \[ p(x) = 2^{-\lceil \log N \rceil} \] uniform distribution.

Aside: Variable-length representations
Unary coding

- If $S = \mathbb{N}$, one can represent $x \in S$ writing $x$ zeroes followed by a one.
If $S = \mathbb{N}$, one can represent $x \in S$ writing $x$ zeroes followed by a one.

So $l_x = x + 1$, and the intended distribution is

$$p(x) = 2^{-x-1} \text{ geometric distribution of ratio } 1/2.$$
If $S = \mathbb{N}$, one can represent $x \in S$ writing $x$ zeroes followed by a one.

So $l_x = x + 1$, and the intended distribution is

\[ p(x) = 2^{-x-1} \]  geometric distribution of ratio 1/2.

\begin{array}{c|c}
0 & 1 \\
1 & 01 \\
2 & 001 \\
3 & 0001 \\
4 & 00001 \\
\end{array}
Unary coding can be seen as a special case of a more general kind of coding for \( \mathbb{N} \). Suppose you group \( \mathbb{N} \) into \textit{slots}: every slot is made by consecutive integers; let

\[ V = \langle s_1, s_2, s_3, \ldots \rangle \]

be the slot sizes (in the unary case \( s_1 = s_2 = \cdots = 1 \)).
Unary coding can be seen as a special case of a more general kind of coding for $\mathbb{N}$. Suppose you group $\mathbb{N}$ into slots: every slot is made by consecutive integers; let

$$V = \langle s_1, s_2, s_3, \ldots \rangle$$

be the slot sizes (in the unary case $s_1 = s_2 = \cdots = 1$). Then, to represent $x \in \mathbb{N}$ one can
A more general viewpoint

Unary coding can be seen as a special case of a more general kind of coding for $\mathbb{N}$. Suppose you group $\mathbb{N}$ into slots: every slot is made by consecutive integers; let

$$V = \langle s_1, s_2, s_3, \ldots \rangle$$

be the slot sizes (in the unary case $s_1 = s_2 = \cdots = 1$). Then, to represent $x \in \mathbb{N}$ one can

- encode *in unary* the index $i$ of the slot containing $x$;
Unary coding can be seen as a special case of a more general kind of coding for $\mathbb{N}$. Suppose you group $\mathbb{N}$ into *slots*: every slot is made by consecutive integers; let

$$V = \langle s_1, s_2, s_3, \ldots \rangle$$

be the slot sizes (in the unary case $s_1 = s_2 = \cdots = 1$). Then, to represent $x \in \mathbb{N}$ one can

- encode *in unary* the index $i$ of the slot containing $x$;
- encode *in binary* the offset of $x$ within its slot (using $\lceil \log s_i \rceil$ bits).
Golomb coding with modulus $b$ is obtained choosing 

\[ V = \langle b, b, b, \ldots \rangle. \]

To represent $x \in \mathbb{N}$ you need to specify the slot where $x$ falls (that is, $\lfloor x/b \rfloor$) in unary, and then represent the offset using $\lceil \log b \rceil$ bits (or $\lfloor \log b \rfloor$ bits).
Golomb coding with modulus \( b \) is obtained choosing

\[ V = \langle b, b, b, \ldots \rangle. \]

To represent \( x \in \mathbb{N} \) you need to specify the slot where \( x \) falls (that is, \( \lfloor x/b \rfloor \)) in unary, and then represent the offset using \( \lceil \log b \rceil \) bits (or \( \lfloor \log b \rfloor \) bits).

So

\[ l_x = \left\lfloor \frac{x}{b} \right\rfloor + \lceil \log b \rceil. \]

The intended distribution is

\[ p(x) = 2^{-l_x} \propto (2^{1/b})^{-x} \quad \text{geometric distribution of ratio } 1/\sqrt{2}. \]
A finer analysis shows that Golomb coding is optimal (=Huffman) for a geometric distribution of ratio $p$, provided that $b$ is chosen as

$$b = \left\lceil \frac{\log(2 - p)}{-\log(1 - p)} \right\rceil.$$
A finer analysis shows that Golomb coding is optimal (\(\equiv\) Huffman) for a geometric distribution of ratio \(p\), provided that \(b\) is chosen as

\[
b = \left\lceil \frac{\log(2 - p)}{-\log(1 - p)} \right\rceil.
\]

Aside: Variable-length representations
Elias’ $\gamma$ coding of $x \in \mathbb{N}^+$ is obtained by representing $x$ in binary preceded by a unary representation of its length (minus one).
Elias’ $\gamma$ coding of $x \in \mathbb{N}^+$ is obtained by representing $x$ in binary preceded by a unary representation of its length (minus one). More precisely, to represent $x$ we write in unary $\lceil \log x \rceil$ and then in binary $x - 2^{\lceil \log x \rceil}$ (on $\lfloor \log x \rfloor$ bits).
Elias’ $\gamma$ coding of $x \in \mathbb{N}^+$ is obtained by representing $x$ in binary preceded by a unary representation of its length (minus one). More precisely, to represent $x$ we write in unary $\lfloor \log x \rfloor$ and then in binary $x - 2^{\lfloor \log x \rfloor}$ (on $\lfloor \log x \rfloor$ bits). So

$$l_x = 1 + 2\lfloor \log x \rfloor \implies p(x) \propto \frac{1}{2x^2} \text{(Zipf of exponent 2)}$$
Elias’ $\gamma$ coding of $x \in \mathbb{N}^+$ is obtained by representing $x$ in binary preceded by a unary representation of its length (minus one). More precisely, to represent $x$ we write in unary $\lfloor \log x \rfloor$ and then in binary $x - 2^{\lfloor \log x \rfloor}$ (on $\lfloor \log x \rfloor$ bits). So

$$l_x = 1 + 2 \lfloor \log x \rfloor \implies p(x) \propto \frac{1}{2x^2} \text{ (Zipf of exponent 2)}$$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>00100</td>
</tr>
<tr>
<td>5</td>
<td>00101</td>
</tr>
</tbody>
</table>
Given an instantaneous code $c$ for the integers, we say that it is universal iff $E_p[c] / H[p]$ is bounded above by a constant for every non-increasing distribution $p$. In other words, a universal code is one that does not loose more than a constant factor with respect to the optimal code independently from the distribution (provided that it is non-increasing). Elias’ $\gamma$ is the first example we meet of a universal code!
Given an instantaneous code $c$ for the integers, we say that it is universal iff $E_p[c]/H[p]$ is bounded above by a constant for every non-increasing distribution $p$. Elias' $\gamma$ is the first example we meet of a universal code!
Universal codes

- Given an instantaneous code $c$ for the integers, we say that it is universal iff $E_p[c]/H[p]$ is bounded above by a constant for every non-increasing distribution $p$.
- In other words, a universal code is one that does not lose more than a constant factor with respect to the optimal code independently from the distribution $p$ (provided that it is non-increasing).

Aside: Variable-length representations
Given an instantaneous code $c$ for the integers, we say that it is universal iff $E_p[c]/H[p]$ is bounded above by a constant for every non-increasing distribution $p$.

In other words, a universal code is one that does not lose more than a constant factor with respect to the optimal code independently from the distribution (provided that it is non-increasing).

Elias’ $\gamma$ is the first example we meet of a universal code!
Elias’ $\delta$ coding of $x \in \mathbb{N}^+$ is obtained by representing $x$ in binary preceded by a representation of its length in $\gamma$. [Also $\delta$ is universal!]

Aside: Variable-length representations
Elias’ δ coding of $x \in \mathbb{N}^+$ is obtained by representing $x$ in binary preceded by a representation of its length in $\gamma$. [Also $\delta$ is universal!]

So

$$l_x = 1 + 2\lfloor \log \log x \rfloor + \lfloor \log x \rfloor \implies p(x) \propto \frac{1}{2x(\log x)^2}$$
Elias’ δ coding of $x \in \mathbb{N}^+$ is obtained by representing $x$ in binary preceded by a representation of its length in γ. [Also δ is universal!]

So

$$l_x = 1 + 2\lfloor \log \log x \rfloor + \lfloor \log x \rfloor \implies p(x) \propto \frac{1}{2x(\log x)^2}$$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0100</td>
</tr>
<tr>
<td>3</td>
<td>0101</td>
</tr>
<tr>
<td>4</td>
<td>01100</td>
</tr>
<tr>
<td>5</td>
<td>01101</td>
</tr>
<tr>
<td>6</td>
<td>0010000</td>
</tr>
<tr>
<td>7</td>
<td>00100001</td>
</tr>
</tbody>
</table>

Aside: Variable-length representations
An alternative way...

...to think of $\gamma$ coding is that $x$ is represented using its usual binary representation (except for the initial “1”, which is omitted), with every bit “coming with” a continuation bit, that tells whether the representation continues or whether it stops there. For example (up to bit permutation) $\gamma$ coding of 724 (in binary: 1011010100) is

```
0 1 1 1 1 1 0 1 1 1 0 1 1 1 1 0 1 0 0
```

Aside: Variable-length representations
What happens if we group digits $k$ by $k$?

$$0111110111110110100$$

$$001111011011000$$

$$01110111011000$$

$$1101101000$$
For $x$, we use $\lceil \log(x)/k \rceil$ bits for the unary part, and the same number of bits multiplied by $k$ for the binary part.
For $x$, we use $\lceil \log(x)/k \rceil$ bits for the unary part, and the same number of bits multiplied by $k$ for the binary part. So

$$l_x = (k+1)(\lceil \log(x)/k \rceil) \quad \implies \quad p(x) \propto x^{-(k+1)/k} (\text{Zipf } (k+1)/k)$$
For $x$, we use $\lceil \log(x)/k \rceil$ bits for the unary part, and the same number of bits multiplied by $k$ for the binary part. So

$$l_x = (k+1)(\lceil \log(x)/k \rceil) \implies p(x) \propto x^{-(k+1)/k} (\text{Zipf} (k+1)/k)$$

A more efficient variant: the $\zeta_k$ codes (for Zipf 1 → 2).

<table>
<thead>
<tr>
<th>$\gamma = \zeta_1$</th>
<th>$\zeta_2$</th>
<th>$\zeta_3$</th>
<th>$\zeta_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
<td>10</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>2 010</td>
<td>110</td>
<td>1010</td>
<td>10010</td>
</tr>
<tr>
<td>3 011</td>
<td>111</td>
<td>1011</td>
<td>10011</td>
</tr>
<tr>
<td>4 00100</td>
<td>01000</td>
<td>1100</td>
<td>10100</td>
</tr>
<tr>
<td>5 00101</td>
<td>01001</td>
<td>1101</td>
<td>10101</td>
</tr>
<tr>
<td>6 00110</td>
<td>01010</td>
<td>1110</td>
<td>10110</td>
</tr>
<tr>
<td>7 00111</td>
<td>01011</td>
<td>1111</td>
<td>10111</td>
</tr>
<tr>
<td>8 0001000</td>
<td>011000</td>
<td>0100000</td>
<td>11000</td>
</tr>
</tbody>
</table>
Comparing codings

Legend
- Unaria = Golomb 1
- Golomb 3
- gamma=1-var
- 3-var
- delta

Aside: Variable-length representations
Graph compression with instantaneous codes: BVGraph
Coding techniques... 

...alone do not improve on compression: we have first to guarantee that the data we represent have a distribution close to the intended one (depending on the coding we are going to use). In particular, they have to enjoy a monotonic distribution (smaller values are more probable than larger ones).
... alone do not improve on compression: we have first to guarantee that the data we represent have a distribution close to the intended one (depending on the coding we are going to use). In particular, they have to enjoy a monotonic distribution (smaller values are more probable than larger ones).

- BTW: some codings (e.g., Elias $\gamma$ and $\delta$) are universal: for whatever monotonic distribution, they guarantee an expected length that is only within a constant factor of the optimal one.
Coding techniques...

...alone do not improve on compression: we have first to guarantee that the data we represent have a distribution close to the intended one (depending on the coding we are going to use). In particular, they have to enjoy a monotonic distribution (smaller values are more probable than larger ones).

- BTW: some codings (e.g., Elias $\gamma$ and $\delta$) are universal: for whatever monotonic distribution, they guarantee an expected length that is only within a constant factor of the optimal one.
- Degrees are often distributed like a Zipf of exponent $\approx 2.7$: they can be safely encoded using $\gamma$. 

Graph compression with instantaneous codes: BVGraph
...alone do not improve on compression: we have first to guarantee that the data we represent have a distribution close to the intended one (depending on the coding we are going to use). In particular, they have to enjoy a monotonic distribution (smaller values are more probable than larger ones).

- BTW: some codings (e.g., Elias $\gamma$ and $\delta$) are universal: for whatever monotonic distribution, they guarantee an expected length that is only within a constant factor of the optimal one.
- Degrees are often distributed like a Zipf of exponent $\approx 2.7$: they can be safely encoded using $\gamma$.
- What about successors? Let us assume that successors of $x$ are $y_1, \ldots, y_k$: how should we encode $y_1, \ldots, y_k$?
Locality

In general, we cannot say much about their distribution, unless we make some assumption on the way in which nodes are numbered.
In general, we cannot say much about their distribution, unless we make some assumption on the way in which nodes are numbered. The following considerations hold true for web graphs!
In general, we cannot say much about their distribution, unless we make some assumption on the way in which nodes are numbered. **The following considerations hold true for web graphs!**

- Many hypertextual links contained in a web page are *navigational* (“home”, “next”, “up”…). If we compare the URL they refer to with that of the page containing them, they share a long common prefix. This property is known as *locality* and it was first observed by the authors of the Connectivity Server.
In general, we cannot say much about their distribution, unless we make some assumption on the way in which nodes are numbered. The following considerations hold true for web graphs!

- Many hypertextual links contained in a web page are *navigational* ("home", "next", "up"...). If we compare the URL they refer to with that of the page containing them, they share a long common prefix. This property is known as *locality* and it was first observed by the authors of the Connectivity Server.

- To exploit this property, assume that URLs are ordered lexicographically (that is, node 0 is the first URL in lexicographic order, etc.). Then, if $x \rightarrow y$ is an arc, most of the times $|x - y|$ will be "small".
If $x$ has successors $y_1 < y_2 < \cdots < y_k$, we represent its successor list though the gaps (differentiation):

$$y_1 - x, y_2 - y_1 - 1, \ldots, y_k - y_{k-1} - 1$$

(only the first value can be negative: we make it into a natural number...). How are such differences distributed?

Zipf with exponent 1.2 $\implies$ we use $\zeta_3$. 
URLs close to each other (in lexicographic order) have similar successor sets: this fact (known as similarity) was exploited for the first time in the Link database.
URLs close to each other (in lexicographic order) have similar successor sets: this fact (known as similarity) was exploited for the first time in the Link database. We may encode the successor list of $x$ as follows:
Similarity

URLs close to each other (in lexicographic order) have similar successor sets: this fact (known as similarity) was exploited for the first time in the Link database. We may encode the successor list of $x$ as follows:

- we write the differences with respect to the successor list of some previous node $x - r$ (called the reference node)
Similarity

URLs close to each other (in lexicographic order) have similar successor sets: this fact (known as similarity) was exploited for the first time in the Link database.

We may encode the successor list of $x$ as follows:

- we write the differences with respect to the successor list of some previous node $x - r$ (called the reference node)
- we explicitly encode (as before) only the successors of $x$ that were not successors of $x - r$. 

Graph compression with instantaneous codes: BVGraph
More explicitly, the successor list of $x$ is encoded as (referencing):

- an integer $r$ (reference): if $r > 0$, the list is described by difference with respect to the successor list of $x - r$; in this case, we write a bitvector (of length equal to $d^+(x - r)$) discriminating the elements in $N^+(x - r) \cap N^+(x)$ from the ones in $N^+(x - r) \setminus N^+(x)$.
More explicitly, the successor list of $x$ is encoded as \textit{(referencing)}:

- an integer $r$ (reference): if $r > 0$, the list is described by difference with respect to the successor list of $x - r$; in this case, we write a bitvector (of length equal to $d^+(x - r)$) discriminating the elements in $N^+(x - r) \cap N^+(x)$ from the ones in $N^+(x - r) \setminus N^+(x)$.
- an explicit list of \textit{extra nodes}, containing the elements of $N^+(x) \setminus N^+(x - r)$ (or the whole $N^+(x)$, if $r = 0$), encoded as explained before.
## Referencing example

Graph compression with instantaneous codes: BVGraph

<table>
<thead>
<tr>
<th>Node</th>
<th>Outdegree</th>
<th>Successors</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>15, 16, 17, 22, 23, 24, 315, 316, 317, 3041</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>13, 15, 16, 17, 50</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Outd.</th>
<th>Ref.</th>
<th>Copy list</th>
<th>Extra nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>0</td>
<td>...</td>
<td>13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>1</td>
<td>01110011010</td>
<td>22, 316, 317, 3041</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>3</td>
<td>11110000000</td>
<td>50</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Instead of using a bitvector, we use run-length encoding, telling the length of successive runs (blocks) of “0” and “1”:

<table>
<thead>
<tr>
<th>Node</th>
<th>Outd.</th>
<th>Ref.</th>
<th>Copy list</th>
<th>Extra nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>0</td>
<td>...</td>
<td>13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>1</td>
<td>01110011010</td>
<td>22, 316, 317, 3041</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>3</td>
<td>111100000000</td>
<td>50</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Instead of using a bitvector, we use run-length encoding, telling the length of successive runs (blocks) of “0” and “1”:

<table>
<thead>
<tr>
<th>Node</th>
<th>Outd.</th>
<th>Ref.</th>
<th>Copy list</th>
<th>Extra nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>0</td>
<td>…</td>
<td>13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>1</td>
<td>01110011010</td>
<td>22, 316, 317, 3041</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>3</td>
<td>11110000000</td>
<td>50</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Outd.</th>
<th>Ref.</th>
<th># blocks</th>
<th>Copy blocks</th>
<th>Extra nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>0</td>
<td>…</td>
<td>…</td>
<td>13, 15, 16, 17, 18, 19, 23, …</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>1</td>
<td>7</td>
<td>0, 0, 2, 1, 1, 0, 0</td>
<td>22, 316, …</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>50</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>
Among the extra nodes, many happen to sport the \textit{consecutivity} property: they appear in clusters of consecutive integers. This phenomenon, observed empirically, have some possible explanations:
Among the extra nodes, many happen to sport the *consecutivity* property: they appear in clusters of consecutive integers. This phenomenon, observed empirically, have some possible explanations:

- most pages contain groups of navigational links that correspond to a certain hierarchical level of the website, and are often consecutive to one another;
Among the extra nodes, many happen to sport the *consecutivity* property: they appear in clusters of consecutive integers. This phenomenon, observed empirically, have some possible explanations:

- most pages contain groups of navigational links that correspond to a certain hierarchical level of the website, and are often consecutive to one another;
- in the transpose graph, moreover, consecutivity is the dual of similarity with reference 1: when there is a cluster of consecutive pages with many similar links, in the transpose there are intervals of consecutive outgoing links.
To exploit consecutivity, we use a special representation for the extra node list called *intervalization*, that is:

- sufficiently long (say $\geq T$) intervals of consecutive integers are represented by their left extreme and their length minus $T$;
- other extra nodes, if any, are called *residual nodes* and are represented alone.
## Intervalization example

<table>
<thead>
<tr>
<th>Node</th>
<th>Outd.</th>
<th>Ref.</th>
<th># blocks</th>
<th>Copy blocks</th>
<th>Extra nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>0</td>
<td>...</td>
<td>...</td>
<td>13, 15, 16, 17, 18, 19, 23, ...</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>1</td>
<td>7</td>
<td>0, 0, 2, 1, 1, 0, 0</td>
<td>22, 316, ...</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>50</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### Intervalization example

<table>
<thead>
<tr>
<th>Node</th>
<th>Outd.</th>
<th>Ref.</th>
<th># blocks</th>
<th>Copy blocks</th>
<th>Extra nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>0</td>
<td>...</td>
<td>...</td>
<td>13, 15, 16, 17, 18, 19, 23, ...</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>1</td>
<td>7</td>
<td>0, 0, 2, 1, 1, 0, 0</td>
<td>22, 316, ...</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>50</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Outd.</th>
<th>Ref.</th>
<th># bl.</th>
<th>Copy bl.s</th>
<th># int.</th>
<th>Lft extr.</th>
<th>Lth</th>
<th>Residuals</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>0</td>
<td>2</td>
<td>15,...</td>
<td>4,...</td>
<td>13, 23 ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>1</td>
<td>7</td>
<td>0, 0, ...</td>
<td>1</td>
<td>316</td>
<td>1</td>
<td>22, 3041</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
When the reference node is chosen, how far back in the “past” are we allowed to go?
When the reference node is chosen, how far back in the “past” are we allowed to go? We need to keep track of a window of the last $W$ successor lists. The choice of $W$ is critical:
When the reference node is chosen, how far back in the “past” are we allowed to go? We need to keep track of a window of the last $W$ successor lists. The choice of $W$ is critical:

- a large $W$ guarantees better compression, but increases compression time and space
When the reference node is chosen, how far back in the “past” are we allowed to go? We need to keep track of a window of the last $W$ successor lists. The choice of $W$ is critical:

- A large $W$ guarantees better compression, but increases compression time and space.
- After $W = 7$ there is no significant improvement in compression.
When the reference node is chosen, how far back in the “past” are we allowed to go? We need to keep track of a window of the last $W$ successor lists. The choice of $W$ is critical:

- a large $W$ guarantees better compression, but increases compression time and space
- after $W = 7$ there is no significant improvement in compression.

The choice of $W$ does not impact on decompression time.
Reference chain length

Referencing involves recursion: to decode the successor list of $x$, we need first to decompress the successor list of $x - r$, etc. This chain is called the *reference chain* of $x$: decompression speed depends on the length of such chains.
Referencing involves recursion: to decode the successor list of $x$, we need first to decompress the successor list of $x - r$, etc. This chain is called the *reference chain* of $x$: decompression speed depends on the length of such chains. During compression, it is possible to limit their length keeping into account of how long is the reference chain for every node in the window and avoiding to use nodes whose reference chain is already of a given maximum length $R$. 
Referencing involves recursion: to decode the successor list of $x$, we need first to decompress the successor list of $x - r$, etc. This chain is called the reference chain of $x$: decompression speed depends on the length of such chains.

During compression, it is possible to limit their length keeping into account of how long is the reference chain for every node in the window and avoiding to use nodes whose reference chain is already of a given maximum length $R$.

The choice of $R$ influences the compression ratio (with $R = \infty$ giving the best possible compression) but also on decompression speed ($R = \infty$ may produce access time that can be two orders of magnitude larger than $R = 1$ — it may even produce stack overflows).
BVGraph for general graphs: LLPA
The basic property we have been exploiting so far is that nodes are numbered according to the lexicographic ordering of URLs. Is it possible to adapt / extend this idea to non-web graphs, e.g., to social networks?
The basic property we have been exploiting so far is that nodes are numbered according to the lexicographic ordering of URLs. Is it possible to adapt / extend this idea to non-web graphs, e.g., to social networks?

What we want is an ordering of the nodes that is compression friendly.
The basic property we have been exploiting so far is that nodes are numbered according to the lexicographic ordering of URLs. Is it possible to adapt / extend this idea to non-web graphs, e.g., to social networks?

- What we want is an ordering of the nodes that is compression friendly
- In particular, we want that most arcs are between nodes that are very close (as numbers) to each other.
Social networks have no natural ordering such as “lexicographic by URL”. However many statistics suggest that social networks are clustered.
Social networks have no natural ordering such as “lexicographic by URL”. However many statistics suggest that social networks are clustered.

Goal: unravel the clustered structure inside social networks, search for an ordering that run through clusters and use it to compress the graph much better than the theoretical lower bound.
Orderings and communities

Social networks have no natural ordering such as “lexicographic by URL”. However many statistics suggest that social networks are clustered.

Goal: unravel the clustered structure inside social networks, search for an ordering that run through clusters and use it to compress the graph much better than the theoretical lower bound.

Constraints:
Social networks have no natural ordering such as “lexicographic by URL”. However many statistics suggest that social networks are clustered.

Goal: unravel the clustered structure inside social networks, search for an ordering that run through clusters and use it to compress the graph much better than the theoretical lower bound.

Constraints:

1. very few clustering techniques scale up to very large graphs
Social networks have no natural ordering such as “lexicographic by URL”. However many statistics suggest that social networks are clustered.

Goal: unravel the clustered structure inside social networks, search for an ordering that run through clusters and use it to compress the graph much better than the theoretical lower bound.

Constraints:

1. very few clustering techniques scale up to very large graphs
2. we do not posses any prior information on the number of clusters
Orderings and communities

Social networks have no natural ordering such as “lexicographic by URL”. However many statistics suggest that social networks are clustered.

Goal: unravel the clustered structure inside social networks, search for an ordering that run through clusters and use it to compress the graph much better than the theoretical lower bound.

Constraints:

1. very few clustering techniques scale up to very large graphs
2. we do not posses any prior information on the number of clusters
3. cluster sizes are going to be very unbalanced
You can obtain an ordering from a clustering just sorting by cluster label. Different clustering algorithms yield different and incomparable orderings. Main idea: Run a clustering algorithm $A$. Renumber nodes sorting by $A$'s labels, breaking ties using the node numbers (i.e., sort stably by $A$'s labels). Iterate with another clustering algorithm.

BVGraph for general graphs: LLPA
You can obtain an ordering from a clustering just sorting by cluster label.
You can obtain an ordering from a clustering just sorting by cluster label.

Different clustering algorithms yield different and incomparable orderings.
You can obtain an ordering from a clustering just sorting by cluster label.

Different clustering algorithms yield different and incomparable orderings.

Main idea:
You can obtain an ordering from a clustering just sorting by cluster label.

Different clustering algorithms yield different and incomparable orderings.

Main idea:
  - Run a clustering algorithm $A$
You can obtain an ordering from a clustering just sorting by cluster label.

Different clustering algorithms yield different and incomparable orderings.

Main idea:
- Run a clustering algorithm $A$
- Renumber nodes sorting by $A$’s labels, breaking ties using the node numbers (i.e., sort stably by $A$’s labels)
You can obtain an ordering from a clustering just sorting by cluster label.

Different clustering algorithms yield different and incomparable orderings.

Main idea:
- Run a clustering algorithm $A$
- Renumber nodes sorting by $A$’s labels, breaking ties using the node numbers (i.e., sort stably by $A$’s labels)
- Iterate with another clustering algorithm

BVGraph for general graphs: LLPA
LPA are a class of clustering algorithm that work as follows:
LPA are a class of clustering algorithm that work as follows:

- Every node adopts the label that is most common among its neighbors...
LPA are a class of clustering algorithm that work as follows:

- Every node adopts the label that is most common among its neighbors...
- ...with an adjustment depending on the overall popularity of the label
Label Propagation Algorithm (LPA)

**Require:**  \( G \) a graph, \( \gamma \) a density parameter
1: \( \pi \leftarrow \) a random permutation of \( G \)'s nodes
2: for all \( x \): \( \lambda(x) \leftarrow x \), \( v(x) \leftarrow 1 \)
3: while (some stopping criterion) do
4:   for \( i = 0, 1, \ldots, n - 1 \) do
5:     for every label \( \ell \), \( k_\ell \leftarrow |\lambda^{-1}(\ell) \cap N_G(\pi(i))| \)
6:     \( \hat{\ell} \leftarrow \text{argmax}_{\ell}[k_\ell - \gamma(v(\ell) - k_\ell)] \)
7:     decrement \( v(\lambda(\pi(i))) \)
8:     \( \lambda(\pi(i)) \leftarrow \hat{\ell} \)
9:     increment \( v(\lambda(\pi(i))) \)
10: end for
11: end while

Here \( v(\ell) \) is the number of nodes currently labelled by \( \ell \), so \( v(\ell) - k_\ell \) is the popularity of label \( \ell \) outside of the current neighborhood.
Layered Label Propagation Algorithm (LLPA)

Repeatedly run LPA with different values of $\gamma$

<table>
<thead>
<tr>
<th>Name</th>
<th>LLP</th>
<th>BFS</th>
<th>Shingle</th>
<th>Natural</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon</td>
<td>9.16 (-30%)</td>
<td>12.96</td>
<td>14.43 (+11%)</td>
<td>16.92 (+30%)</td>
<td>23.62 (+82%)</td>
</tr>
<tr>
<td>DBLP</td>
<td>6.88 (-23%)</td>
<td>8.91</td>
<td>11.42 (+28%)</td>
<td>11.36 (+27%)</td>
<td>22.07 (+147%)</td>
</tr>
<tr>
<td>Enron</td>
<td>6.51 (-24%)</td>
<td>8.54</td>
<td>9.87 (+15%)</td>
<td>13.43 (+57%)</td>
<td>14.02 (+64%)</td>
</tr>
<tr>
<td>Hollywood</td>
<td>5.14 (-35%)</td>
<td>7.81</td>
<td>6.72 (-14%)</td>
<td>15.20 (+94%)</td>
<td>16.23 (+107%)</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>10.90 (-28%)</td>
<td>15.1</td>
<td>15.77 (+4%)</td>
<td>14.35 (-5%)</td>
<td>23.50 (+55%)</td>
</tr>
<tr>
<td>Flickr</td>
<td>8.89 (-22%)</td>
<td>11.26</td>
<td>10.22 (-10%)</td>
<td>13.87 (+23%)</td>
<td>14.49 (+28%)</td>
</tr>
<tr>
<td>indochina (hosts)</td>
<td>5.53 (-17%)</td>
<td>6.63</td>
<td>7.16 (+7%)</td>
<td>9.26 (+39%)</td>
<td>10.59 (+59%)</td>
</tr>
<tr>
<td>uk (hosts)</td>
<td>6.26 (-18%)</td>
<td>7.62</td>
<td>8.12 (+6%)</td>
<td>10.81 (+41%)</td>
<td>15.58 (+104%)</td>
</tr>
<tr>
<td>eu</td>
<td>3.90 (-21%)</td>
<td>4.93</td>
<td>6.86 (+39%)</td>
<td>5.24 (+6%)</td>
<td>19.89 (+303%)</td>
</tr>
<tr>
<td>in</td>
<td>2.46 (-30%)</td>
<td>3.51</td>
<td>4.79 (+36%)</td>
<td>2.99 (-15%)</td>
<td>21.15 (+502%)</td>
</tr>
<tr>
<td>indochina</td>
<td>1.71 (-26%)</td>
<td>2.31</td>
<td>3.59 (+55%)</td>
<td>2.19 (-6%)</td>
<td>21.46 (+829%)</td>
</tr>
<tr>
<td>it</td>
<td>2.10 (-28%)</td>
<td>2.89</td>
<td>4.39 (+51%)</td>
<td>2.83 (-3%)</td>
<td>26.40 (+813%)</td>
</tr>
<tr>
<td>uk</td>
<td>1.91 (-33%)</td>
<td>2.84</td>
<td>4.09 (+44%)</td>
<td>2.75 (-4%)</td>
<td>27.55 (+870%)</td>
</tr>
<tr>
<td>altavista-nd</td>
<td>5.22 (-11%)</td>
<td>5.85</td>
<td>8.12 (+38%)</td>
<td>8.37 (+43%)</td>
<td>34.76 (+494%)</td>
</tr>
</tbody>
</table>
Layered Label Propagation Algorithm (LLPA)

- Repeatedly run LPA with different values of $\gamma$
Layered Label Propagation Algorithm (LLPA)

- Repeatedly run LPA with different values of $\gamma$
- Renumber nodes sorting by stably by the new labels

<table>
<thead>
<tr>
<th>Name</th>
<th>LLP</th>
<th>BFS</th>
<th>Shingle</th>
<th>Natural</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon</td>
<td>9.16</td>
<td>12.96</td>
<td>14.43 (+11%)</td>
<td>16.92 (+30%)</td>
<td>23.62 (+82%)</td>
</tr>
<tr>
<td>DBLP</td>
<td>6.88</td>
<td>8.91</td>
<td>11.42 (+28%)</td>
<td>11.36 (+27%)</td>
<td>22.07 (+147%)</td>
</tr>
<tr>
<td>Enron</td>
<td>6.51</td>
<td>8.54</td>
<td>9.87 (+15%)</td>
<td>13.43 (+57%)</td>
<td>14.02 (+64%)</td>
</tr>
<tr>
<td>Hollywood</td>
<td>5.14</td>
<td>7.81</td>
<td>6.72 (-14%)</td>
<td>15.20 (+94%)</td>
<td>16.23 (+107%)</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>10.90</td>
<td>15.1</td>
<td>15.77 (+4%)</td>
<td>14.35 (-5%)</td>
<td>23.50 (+55%)</td>
</tr>
<tr>
<td>Flickr</td>
<td>8.89</td>
<td>11.26</td>
<td>10.22 (-10%)</td>
<td>13.87 (+23%)</td>
<td>14.49 (+28%)</td>
</tr>
<tr>
<td>indochina (hosts)</td>
<td>5.53</td>
<td>6.63</td>
<td>7.16 (+7%)</td>
<td>9.26 (+39%)</td>
<td>10.59 (+59%)</td>
</tr>
<tr>
<td>uk (hosts)</td>
<td>6.26</td>
<td>7.62</td>
<td>8.12 (+6%)</td>
<td>10.81 (+41%)</td>
<td>15.58 (+104%)</td>
</tr>
<tr>
<td>eu</td>
<td>3.90</td>
<td>4.93</td>
<td>6.86 (+39%)</td>
<td>5.24 (+6%)</td>
<td>19.89 (+303%)</td>
</tr>
<tr>
<td>in</td>
<td>2.46</td>
<td>3.51</td>
<td>4.79 (+36%)</td>
<td>2.99 (-15%)</td>
<td>21.15 (+502%)</td>
</tr>
<tr>
<td>indochina</td>
<td>1.71</td>
<td>2.31</td>
<td>3.59 (+55%)</td>
<td>2.19 (-6%)</td>
<td>21.46 (+829%)</td>
</tr>
<tr>
<td>it</td>
<td>2.10</td>
<td>2.89</td>
<td>4.39 (+51%)</td>
<td>2.83 (-3%)</td>
<td>26.40 (+813%)</td>
</tr>
<tr>
<td>uk</td>
<td>1.91</td>
<td>2.84</td>
<td>4.09 (+44%)</td>
<td>2.75 (-4%)</td>
<td>27.55 (+870%)</td>
</tr>
<tr>
<td>altavista-nd</td>
<td>5.22</td>
<td>5.85</td>
<td>8.12 (+38%)</td>
<td>8.37 (+43%)</td>
<td>34.76 (+494%)</td>
</tr>
</tbody>
</table>
Graph compression with Elias-Fano: EFGraph
Elias-Fano representation

Elias proposed in 1975 a general representation for monotone sequences, later discussed by Fano. In 2012 Vigna proposed to use it for inverted indices, and in particular for storing successor lists in graph compression. A very general technique!

Graph compression with Elias-Fano: EFGraph
Elias proposed in 1975 a *general* representation for monotone sequences, later discussed by Fano.
Elias proposed in 1975 a general representation for monotone sequences, later discussed by Fano.

In 2012 Vigna proposed to use it for inverted indices, and in particular for storing successor lists in graph compression...
Elias proposed in 1975 a *general* representation for monotone sequences, later discussed by Fano.

In 2012 Vigna proposed to use it for inverted indices, and in particular for storing successor lists in graph compression...

A very general technique!
Given a non-decreasing sequence:

\[ 0 \leq x_1, \ldots, x_d < n \]

(e.g., the successors of a node of degree \( d \) in a graph of \( n \) nodes):
Given a non-decreasing sequence:

\[ 0 \leq x_1, \ldots, x_d < n \]

(e.g., the successors of a node of degree \( d \) in a graph of \( n \) nodes):

- let \( \ell = \lceil \log(n/d) \rceil \)
Given a non-decreasing sequence:

\[ 0 \leq x_1, \ldots, x_d < n \]

(e.g., the successors of a node of degree \( d \) in a graph of \( n \) nodes):

- let \( \ell = \left\lfloor \log(n/d) \right\rfloor \)
- write the \( \ell \) lower bits of \( x_i \) explicitly
Given a non-decreasing sequence:

\[ 0 \leq x_1, \ldots, x_d < n \]

(e.g., the successors of a node of degree \( d \) in a graph of \( n \) nodes):

- let \( \ell = \lfloor \log(n/d) \rfloor \)
- write the \( \ell \) lower bits of \( x_i \) explicitly
- write the upper bits as gaps, written in unary
Given a non-decreasing sequence:

\[ 0 \leq x_1, \ldots, x_d < n \]

(e.g., the successors of a node of degree \( d \) in a graph of \( n \) nodes):

- let \( \ell = \lfloor \log(n/d) \rfloor \)
- write the \( \ell \) lower bits of \( x_i \) explicitly
- write the upper bits as gaps, written in unary

\[ 5, 8, 8, 15, 32 \leq n = 36, \ell = 2 \]
So $x_1 = y_1 \cdot 2^\ell + r_1$, ..., $x_d = y_d \cdot 2^\ell + r_d$, with $r_i$ written using $\ell$ bits each, and $g_1 = y_1 - 0$, $g_2 = y_2 - y_1$, ..., $g_d = y_d - y_{d-1}$ written in unary:
Idea

So $x_1 = y_1 \cdot 2^\ell + r_1, \ldots, x_d = y_d \cdot 2^\ell + r_d$, with $r_i$ written using $\ell$ bits each, and $g_1 = y_1 - 0, g_2 = y_2 - y_1, \ldots, g_d = y_d - y_{d-1}$ written in unary:

- writing the $g_i$’s requires $d$ stopping bits...
So $x_1 = y_1 \cdot 2^\ell + r_1$, $\ldots$, $x_d = y_d \cdot 2^\ell + r_d$, with $r_i$ written using $\ell$ bits each, and $g_1 = y_1 - 0$, $g_2 = y_2 - y_1$, $\ldots$, $g_d = y_d - y_{d-1}$ written in unary:

- writing the $g_i$’s requires $d$ stopping bits…
- $\ldots$ plus $g_1 + \cdots + g_d$ bits
Idea

So $x_1 = y_1 \cdot 2^\ell + r_1$, \ldots, $x_d = y_d \cdot 2^\ell + r_d$, with $r_i$ written using $\ell$ bits each, and $g_1 = y_1 - 0$, $g_2 = y_2 - y_1$, \ldots, $g_d = y_d - y_{d-1}$ written in unary:

- writing the $g_i$’s requires $d$ stopping bits...
- \ldots plus $g_1 + \cdots + g_d$ bits
- $g_1 + \cdots + g_d = y_d \leq n/2^\ell \leq n/2^{\log(n/d) - 1} = 2d$ bits.

All in all we need $2 + \lceil \log(n/d) \rceil$ bits per element.

The representation is almost optimal (Elias proves that it is < 0.5 bit away from the information-theoretic lower bound).
So $x_1 = y_1 \cdot 2^\ell + r_1$, \ldots, $x_d = y_d \cdot 2^\ell + r_d$, with $r_i$ written using $\ell$ bits each, and $g_1 = y_1 - 0, g_2 = y_2 - y_1, \ldots, g_d = y_d - y_{d-1}$ written in unary:

- writing the $g_i$’s requires $d$ stopping bits…
- \ldots plus $g_1 + \cdots + g_d$ bits
- $g_1 + \cdots + g_d = y_d \leq n/2^\ell \leq n/2^{\log(n/d)-1} = 2d$ bits.

All in all we need $2 + \lceil \log(n/d) \rceil$ bits per element.
Idea

So \( x_1 = y_1 \cdot 2^\ell + r_1, \ldots, x_d = y_d \cdot 2^\ell + r_d \), with \( r_i \) written using \( \ell \) bits each, and \( g_1 = y_1 - 0, g_2 = y_2 - y_1, \ldots, g_d = y_d - y_{d-1} \) written in unary:

- writing the \( g_i \)'s requires \( d \) stopping bits...
- \ldots plus \( g_1 + \cdots + g_d \) bits
- \( g_1 + \cdots + g_d = y_d \leq n/2^\ell \leq n/2^{\log(n/d)-1} = 2d \) bits.

All in all we need \( 2 + \lceil \log(n/d) \rceil \) bits per element.

The representation is almost optimal (Elias proves that it is < .5 bit away from the information-theoretic lower bound).